
Projet de Bases de Données et Conception Objets

# Documentation Finale

*Élèves :*
Allak Saad Ouassil
Chifre Fabien
Dienstmann Felipe
Meireles João Pedro
Schuh Matheus

*Enseignants :*
Altisen Karine
Bobineau Christophe

Avril 2015

# Table des matières

**1 Introduction** **2**

**2 Analyse** **3**
2.1 Analyse dynamique . . . . . 3
2.2 Analyse statique . . . . . 3

**3 Conception de bases de données** **6**
3.1 Élaboration du schéma conceptuel . . . . . 6
3.2 Conception de la base de données . . . . . 7
3.2.1 Traduction du schéma en relationnel . . . . . 7
3.2.2 Tables et Contraintes . . . . . 8
3.2.3 Forme normale des relations . . . . . 9
3.3 Gestion des accès à la base de données . . . . . 12

**4 Analyse de l'application** **13**
4.1 Cas d'utilisation . . . . . 13
4.2 Cas d'utilisation et base de données . . . . . 15

**5 Conception de l'application** **16**
5.1 Architecture de l'application . . . . . 16
5.2 Fonctionnement de la synchronisation avec la BDD . . . . . 17
5.3 Dictionnaire de classes . . . . . 19
5.3.1 Le modèle . . . . . 19
5.3.2 La couche DAO . . . . . 20
5.3.3 La couche vue . . . . . 20
5.3.4 La couche contrôleur . . . . . 20

**6 Bilan** **21**

# Table des figures

1 Le Schéma Entité/Associations de l'application de Bataille Navale . . 6
2 Diagramme de cas d'utilisation . . . . . 13
3 Diagramme de classe . . . . . 17
4 Exemple d'une transaction sur un tour . . . . . 18

# Liste des tableaux

# 1 Introduction

Ce projet est réalisé dans le cadre du cours d'ACVL et de principe des systèmes de bases de données.

Il a pour but de réaliser une application de jeu de bataille navale en équipe.

Il comprend la réalisation de toute les étapes de réalisation : l'analyse, la conception et l'implémentation de la base de données ainsi que de l'application elle même.

Cette application doit être réalisée en Java et doit disposer d'une base de données implémantée dans SQL Plus.

# 2 Analyse

## 2.1 Analyse dynamique

## 2.2 Analyse statique

**Propriétés Élémentaires** En analysant le cahier des charges, avec lectures répétées, annotations et recherche des informations implicites,nous avons trouvé les données élémentaires de l'application résumées ci-dessous :

{

- pseudo, nom, prenom, mail, adrNumero, adrRue, adrCodePostal, adrVille, dateNaissance
- idPartie, dateDemarage, numJoueur
- idBateau, pivotX, pivotY, orientation, pointVie, taille
- numAction, tour, typeDeplacement, cibleTirX, cibleTirY

}

Quelques clarifications :

- **numJoueur** donne le numéro d'un joueur dans une partie, qui peut être soit un, soit deux. Ce numéro est important pour déterminer qui commence à joueur et pour le passage des tours.
- **numAction** ce numéro est utilisé pour représenter l'ordre des actions dans un tour.

**Dépendances Fonctionnelles** Une dépendance fonctionnelle (DF) est un cas particulier de contrainte relationnelle, où soit $R(X, Y, Z)$ un schéma relationnel avec $X, Y, Z$ un ensemble d'attributs ($Z$ peut être vide), $X$ détermine $Y$ (noté $X \rightarrow Y$) si et seulement si :

$$\forall t_1, \forall t_2 \in R \text{ si } t_1[X] = t_2 \Rightarrow t_1[Y] = t_2[Y]$$

Après familiarisation avec le domaine du problème à modéliser, les DF qui suivent ont été extraites du sujet :

- pseudo → (nom, prenom, mail, adrNumero, adrRue, adrCodePostal, adrVille, dateNaissance)
- idPartie → (dateDemarrage)
- (idPartie, pseudo) → (numJoueur)
- (idPartie, idBateau) → (taille, pivotX, pivotY, orientation, pointVie, pseudo)

- (idPartie, tour, numAction) → (idBateau, pseudo)
- (idPartie, tour, numAction) → (typeDeplacement) *[pour les actions de déplacement]*
- (idPartie, tour, numAction) → (cibleTirX, cibleTirY) *[pour les actions de tir]*

**Contraintes de valeur** Concernant les contraintes de valeur, nous avons trouvé les suivantes :

- dateDemarrage $\geq$ une certaine date crédible (après le début de l'année 2015)
- dateNaissance $\geq$ une date crédible (après l'année 1900)
- numero $> 0$
- codePostal $> 0$
- taille $= 2$ ou $3$
- pointVie $\geq 0$
- typeDeplacement $\in \{droite, gauche, avancer, reculer\}$
- orientation $\in \{nord, sud, est, ouest\}$
- pivotX, pivotY, orientation, taille $\Rightarrow$ bateau dans la grille
- cibleTirX, cibleTirY $\Rightarrow$ tir dans la grille
- tour $> 0$
- numAction $> 0$
- numJoueur $= 1$ ou $2$

**Contraintes de multiplicité** Des restrictions sur la multiplicité des associations entre les tables du domaine sont exprimées ici :

- Un joueur a un destroyer par partie
- Un joueur a 1 ou 2 escorteurs par partie
- Dans une partie, il y a de 4 à 6 bateaux (1 destroyer + 1 ou 2 escorteurs par joueur)
- Une partie a 0 (partie non terminée) ou 1 vainqueur (partie terminée)
- Un jouer peut avoir gagné de zéro à plusieurs parties
- Un jouer peut jouer de zéro à plusieurs parties
- Une partie est jouée par 2 joueurs

**Autres Contraintes** D'autres restrictions, pas forcement évidentes pour l'implémentation de la base de données, ont été trouvées avec une analyse plus profonde du sujet :

- Deux bateaux ne peuvent pas se superposer.
- Le vainquer d'une partie l'a joué.

# 3 Conception de bases de données

## 3.1 Élaboration du schéma conceptuel

Le Schéma Entité/Associations, construit à partir de l'analyse statique, est un modèle de représentation abstrait du domaine, essentiel pour la suite de la conception. Il est indépendant, mais compatible, avec tout modèle de donnés utilisé par la BD, en plus il élimine les possibles redondances de concept et même s'il est assez simple, il apporte une grande richesse sémantique.

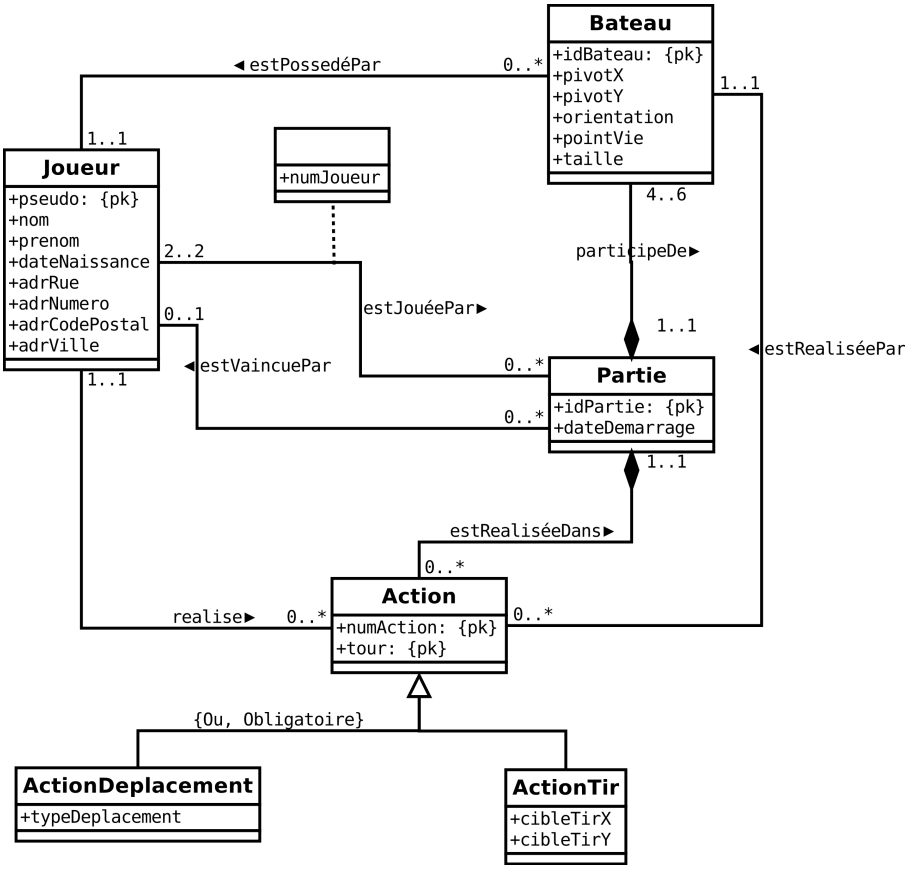


FIGURE 1 – Le Schéma Entité/Associations de l'application de Bataille Navale

**Explication du schéma** Chaque bateau a une taille dans la grille qui va définir son nom dans l'application. Il est lié forcément à un joueur qui le possède et peut faire des actions en l'utilisant.

Une action d'un bateau, réalisé par un joueur qui a un numJoueur (1 ou 2) peut être soit une action de déplacement, soit une action de tir. Ces deux possibilités sont représentées par les sous-types ActionDeplacement et ActionTir de l'entité Action, avec les contraintes adéquates sur l'héritage (Obligatoire, Ou pour partitionner l'ensemble d'actions).

Comme les bateaux et les actions doivent forcement être associés à une partie pour exister, ils sont représentés par les entités faibles Bateau et Action liées à Partie.

**Contraintes non représentées** Le schéma Entités/Associations ne représente pas toutes les contraintes identifiées dans l'analyse statique. Les contraintes de valeur ne sont pas considérées dans cette partie de la conception de la base de données. La plupart des contraintes de multiplicités sont traduites en associations dans le schéma. Cependant, certaines d'entre elles ne sont pas représentées :

- la vérification du nombre de bateaux de chaque type (destroyers et escorteurs) qui appartient à un joueur dans une partie ;
- la validité des actions et des positions des bateaux (deux bateaux ne peuvent pas se superposer, par exemple).

Ces contraintes seront prises en compte plus tard dans la conception de la base de données.

## 3.2 Conception de la base de données

### 3.2.1 Traduction du schéma en relationnel

Pour traduir le schéma qui nous avons faire précédemment, nous avons se utiliser de un algorithme divisé en cinq parties. La description de chaque partie et son resultat sont décrit suivant :

1. Premièrement c'est nécessaire de créer une relation par entité présente dans notre schéma, ainsi que pour ses éventuelles sous-entités. Pour les deux sous-entités de notre application (une pour chaque type de action) nous avons choisi le stratégie de clef étrangère : pour chaque fils on ajoute les clefs primaire de l'entité parente. Ainsi on a crée les entités suivantes : Joueur, Bateau, Partie, Action, ActionDeplacement et ActionTir.

2. Après cette partie, nous commençons a traduire chaque association présente dans le schéma. Les premieres sont des associations qui représentent l'idée de "faiblesse", cette-à-dire, un classe est faible par rapport a autre via l'association. Dans notre diagramme on a deux cas d'entités faibles, Action e Bateau. Pour traduire ceci, on ajoute les clefs de l'entité "forte" (Partie) comme clefs primaires dans ces relations.

3. Maintenant il est nécessaire de traiter les associations unitaires qui ne représentent pas l'idée de faiblesse, en ajoutant les clefs primaires de l'entité qui est unique dans l'association comme clef étrangère dans l'autre. Comme on a deux cas comme cela, les associations entre Action et Partie/Joueur, on ajoute les clefs de Partie et Joueur comme clef étrangère de Action.

4. La quatrième partie du algorithme traduit des associations qui ont une cardinalité (0..1). Pour faire cela on va créer un nouvelle relation avec les attributs de l'association ainsi que les clefs de l'entité à coté du (0..1) comme clefs étrangères et les clefs de l'autre entité comme clefs primaires. On crée donc un nouvelle relation qui s'appele Vainqueurs (qui traduit l'association "estVaincuePar" entre Joueur et Partie)

5. Pour finir on va prendre tous les autres associations, cette-à-dire : les associations de cardinalité (x..*), et on va créer des nouvelles relations avec les attributs de l'association et les clefs des deux entités comme clefs primaires. Cela nous mène à crée un nouvelle relation qui s'appele JoueursParPartie (qui traduit l'association "estJoueuPar" entre Joueur et Partie).

### 3.2.2 Tables et Contraintes

Après la traduction, le schéma relationnel obtenu est le suivant (**gras** représente clé primaire, *italique* représente clé étranger et on peut avoir ***les deux*** ensemble aussi) :

| Bateau | | | | | | | |
|---|---|---|---|---|---|---|---|
| **idBateau** | ***idPartie*** | *pseudo* | taille | pivotX | pivotY | orientation | pointVie |

| Partie | |
|---|---|
| **idPartie** | dateDemarage |

| Joueur | | | | | | | |
|---|---|---|---|---|---|---|---|
| **pseudo** | nom | prenom | dateNaissance | adrRue | adrNumero | adrCodePostal | adrVille |

| Action | | | | |
|---|---|---|---|---|
| ***idPartie*** | **tour** | **numAction** | *idBateau* | *pseudo* |

| ActionDeplacement | | | |
|---|---|---|---|
| ***idPartie*** | **tour** | **numAction** | typeDeplacement |

| ActionTir | | | | |
|---|---|---|---|---|
| ***idPartie*** | **tour** | **numAction** | cibleTirX | cibleTirY |

| Vainqueurs | |
|---|---|
| ***idPartie*** | *pseudo* |

| JoueursParPartie | | |
|---|---|---|
| ***pseudo*** | ***idPartie*** | numJoueur |

**Contraintes d'intégrité en SQL** Toutes les **contraintes** de valeurs ont été facilement **implantées** en SQL, avec l'utilisation de la logique classique, l'algèbre booléenne, `CONSTRAINT` et `CHECK`. La plupart des contraintes de multiplicité ont été implicitement implantées dans le schéma Entité/Association et ont été représentées lors de la traduction en schéma relationnel et conséquemment la création des tables.

Par contre quelques **contraintes** de multiplicités n'ont été **pas possible d'implanter** en SQL et le code Java sera responsable pour vérifier le suivi de ces restrictions :

- Un joueur a un destroyer par partie
- Un joueur a 1 ou 2 escorteurs par partie
- 1 destroyer + 1 ou 2 escorteurs par joueur
- Deux bateaux ne peuvent pas se superposer
- Le vainquer d'une partie l'a joué

### 3.2.3 Forme normale des relations

Pour chaque relation obtenue à partir du schéma entité-association, on vérifie sa forme normale en regardant les dépendances fonctionnelles associées. Pour être plus concis, nous disons ici qu'elles sont toutes en 1ère forme normale parce que tous ses attributs sont de type atomique.

#### Bateau

- **Dépendances fonctionnelles :**
  - (idPartie, idBateau) → taille
  - (idPartie, idBateau) → pivot
  - (idPartie, idBateau) → orientation
  - (idPartie, idBateau) → pointVie
  - (idPartie, idBateau) → pseudo
- **Attributs clefs :** idPartie, idBateau
- **Atributs non-clefs :** taille, pivot, orientation, pointVie, pseudo

Comme, pour chaque attribut non-clef, on a une dépendance fonctionnelle fonctionnelle qui le lie avec la clef {*idPartie, idBateau*}, ils sont tous pleinement dépendants de cette clef, et alors la relation est en 2ème forme normale (**2FN**). Chaque attribut non-clef ne dépend que des attributs clefs, alors la relation est en 3ème forme normale (**3FN**). Toutes les dépendances fonctionnelles contiennent la clef {*idPartie, idBateau*} en partie gauche, alors la relation est en forme normale de Boyce-Codd-Kent (**3FNBCK**).

**Partie**

- **Dépendances fonctionnelles :**
  - idPartie → dateDemarrage
- **Attributs clefs :** idPartie
- **Atributs non-clefs :** dateDemarrage

Le seul attribut non-clef *dateDemarrage* est pleinement dépendant de la clef *idPartie*, alors la relation est en **2FN**. Il ne dépend que du seule attribut clef, alors la relation est en **3FN**. La seule dépendance fonctionnelle contient la clef *idPartie* en partie gauche, alors la relation est en **3FNBCK**.

**Jouer**

- **Dépendances fonctionnelles :**
  - pseudo → nom
  - pseudo → prenom
  - pseudo → dateNaissance
  - pseudo → adrRue
  - pseudo → adrNumero
  - pseudo → adrCodePostal
  - pseudo → adrVille
- **Attributs clefs :** pseudo
- **Atributs non-clefs :** nom, prenom, dateNaissance, adrRue, adrNumero, adrCodePostal, adrVille

Les attributs non-clef sont tous pleinement dépendants de la clef *pseudo*, alors la relation est en **2FN**. Ils ne dépendent que de cet attribut clef, alors la relation est en **3FN**. Les dépendances fonctionnelles contiennent toutes la clef *pseudo* en partie gauche, alors la relation est en **3FNBCK**.

**Action**

- **Dépendances fonctionnelles :**
  - (idPartie, tour, numAction) → idBateau
  - (idPartie, tour, numAction) → pseudo
- **Attributs clefs :** idPartie, tour, numAction
- **Atributs non-clefs :** idBateau, pseudo

Les attributs non-clef sont tous pleinement dépendants de la clef {*idPartie, tour, numAction*}, alors la relation est en **2FN**. Ils ne dépendent que des attributs de la clef, alors la relation est en **3FN**. Les dépendances fonctionnelles contiennent toutes la clef {*idPartie, tour, numAction*} en partie gauche, alors la relation est en **3FNBCK**.

**ActionDeplacement**

- **Dépendances fonctionnelles :**
  - (idPartie, tour, numAction) → typeDeplacement
- **Attributs clefs :** idPartie, tour, numAction
- **Atributs non-clefs :** typeDeplacement

Le seule attribut non-clef (*typeDeplacement*) est pleinement dépendant de la clef {*idPartie, tour, numAction*}, alors la relation est en **2FN**. Ils ne dépendent que des attributs de la clef, alors la relation est en **3FN**. La seule dépendance fonctionnelle contient la clef {*idPartie, tour, numAction*} en partie gauche, alors la relation est en **3FNBCK**.

**ActionTir**

- **Dépendances fonctionnelles :**
  - (idPartie, tour, numAction) → cibleTirX
  - (idPartie, tour, numAction) → cibleTirY
- **Attributs clefs :** idPartie, tour, numAction
- **Atributs non-clefs :** cibleTirX, cibleTirY

Les attributs non-clef sont pleinement dépendants de la clef {*idPartie, tour, numAction*}, alors la relation est en **2FN**. Ils ne dépendent que des attributs de la clef, alors la relation est en **3FN**. Les dépendances fonctionnelles contiennent la clef {*idPartie, tour, numAction*} en partie gauche, alors la relation est en **3FNBCK**.

**Vainqueurs**

- **Dépendances fonctionnelles :**
  - idPartie → pseudo (valide pour cette relation, car pour toutes les parties dans cette table, on connaît son vainqueur)
- **Attributs clefs :** idPartie
- **Atributs non-clefs :** pseudo

Le seul attribut non-clef *pseudo* est pleinement dépendant de la clef *idPartie*, alors la relation est en **2FN**. Il ne dépend que du seule attribut clef, alors la relation est en **3FN**. La seule dépendance fonctionnelle contient la clef *idPartie* en partie gauche, alors la relation est en **3FNBCK**.

**JoueursParPartie**

- **Dépendances fonctionnelles :**
  - (idPartie, pseudo) → numJoueur
- **Attributs clefs :** idPartie, pseudo
- **Atributs non-clefs :** numJoueur

Le seul attribut non-clef *numJoueur* est pleinement dépendant de la clef {*idPartie, pseudo*}, alors la relation est en **2FN**. Il ne dépend que de cette clef, alors la relation est en **3FN**. La seule dépendance fonctionnelle contient la clef {*idPartie, pseudo*} en partie gauche, alors la relation est en **3FNBCK**.

## 3.3 Gestion des accès à la base de données

Cette partie est détaillée dans l'analyse une fois les cas d'utilisation définis.

# 4 Analyse de l'application

La phase d'analyse correspond principalement l'identification des différents cas d'utilisation.

Nous n'avons pas réalisé un diagramme de classe purement d'analyse en nous intéressant directement à la conception.

## 4.1 Cas d'utilisation

Ceux-ci se retrouvent sur le diagramme suivant :

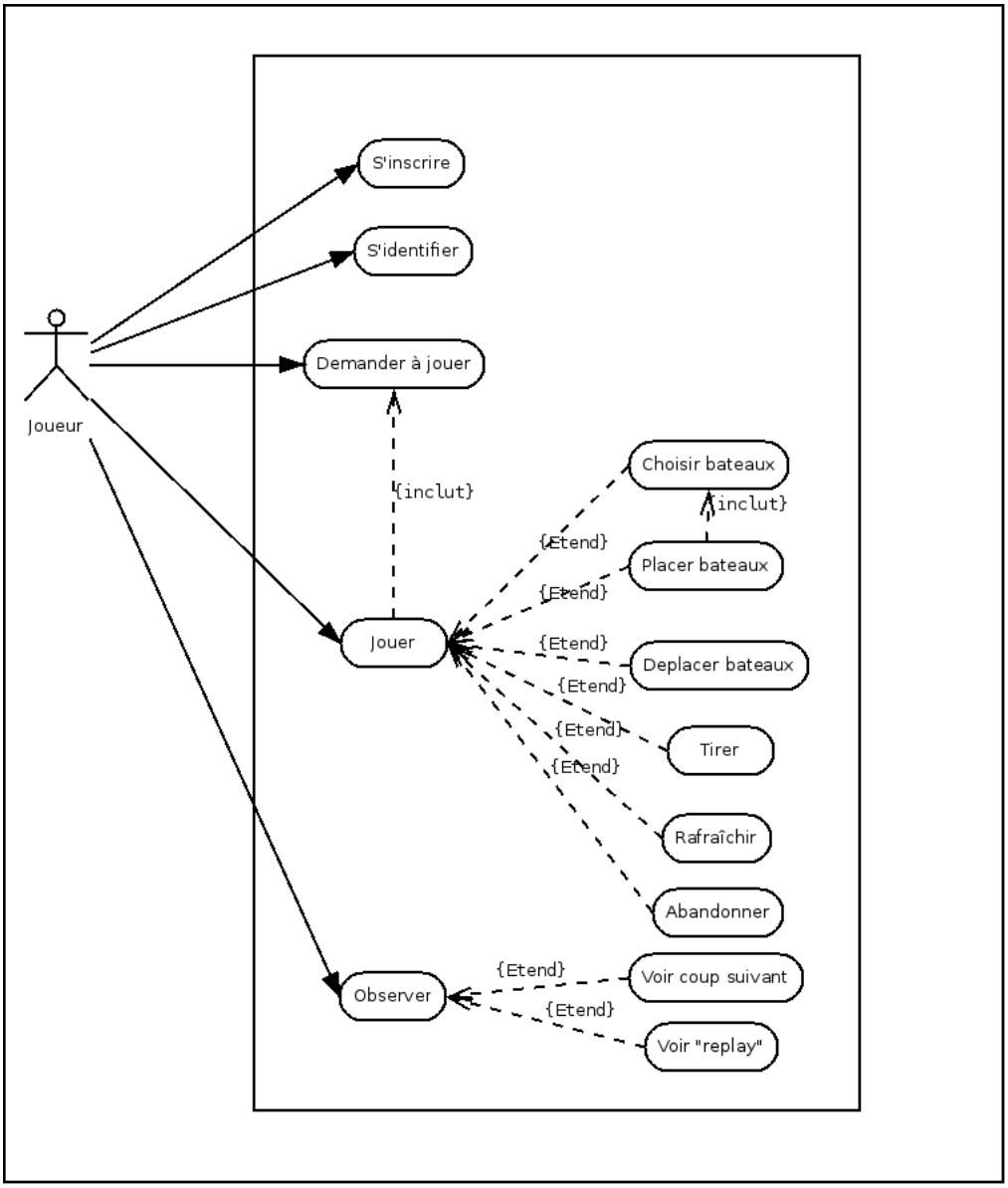


FIGURE 2 – Diagramme de cas d'utilisation

Voici leur description :

- **S'inscrire** :L'utilisateur entre l'ensemble des informations nécessaire à son inscription.Si ces renseignements sont plausible (date de naissance cohérente, numéro de rue supérieur à 0), l'utilisateur est crée. Si le pseudo choisi existe déjà, on invite l'utilisateur à choisir un autre pseudo.

- **S'identifier** : L'utilisateur entre son pseudo puis sa date de naissance. Si les informations correspondent à un joueur inscrit, l'utilisateur est connecté en temps que ce joueur et arrive sur l'écran d'acceuil de l'application. S'il y a une erreur, l'utilisateur en est informé.

- **Demander à jouer** : A partir de l'écran d'acceuil, l'utilisateu peut indiquer qu'il souhaite jouer. Pour ce faire, soit il sélectionne une partie sur lequel il est déjà joueur soit il sélectionne le bouton pour en créer une. Dans les deux cas, il arrive sur l'écran de jeu.

- **Jouer** : Correspond à la phase de jeu de l'application. Chaque joueur joue chacun son tour. Chaque tour dispose d'un nombre d'actions limité.

- **Choisir bateaux** : Avant le début de la partie le joueur doit selectionner les bateaux avec lesquels il souhaite jouer.

- **Placer bateaux** : Une fois les bateaux choisis, le joueur doit les placer sur sa carte. Il indique ensuite qu'il a fini son action.

- **Deplacer bateaux** : Lors de son tour de jeu, le joueur peut choisir de déplacer un bateau. Il selectionne alors le bateau souhaité puis choisi l'action. Il peut avancer, tourner à droite, à gauche ou reculer. Le mouvement doit être réalisable.Cela compte comme une action dans le tour courant et ne peut pas être réalisé si ce nombre est atteint.

- **Tirer** : Lors de son tour de jeu, le joueur peut choisir de tirer. Il selectionne alors le bateau avec lequel il souhaite tirer puis sa cible sur la carte adverse. Cela compte comme une action dans le tour courant et ne peut pas être réalisé si ce nombre est atteint.

- **Abandonner** : Lorsqu'il est sur une partie, un joueur peut décider d'abandoner. La partie est terminée et son adversaire est déclaré vainqueur.

- **Rafraîchir** : Lorsque ce n'est pas son tour de jeu, le joueur peut rafraichir. Si son adversaire a terminé toutes ses actions, le joueur les reçoit et c'est alors à lui de jouer.

- **Observer** : A partir de l'écran d'acceuil un utilisateur voit la liste des parties en cours et terminées et peut choisir d'en observer une.

- **Voir coup suivant** : Lorsqu'il observe une partie, l'utilisateur peut voir le coup suivant en cliquant sur le bouton correspondant. S'il n'y a pas eu de coup entre temps, rien ne se passe.

- **Voir replay** : Lorsqu'il observe une partie, l'utilisateur peut consulter les actions précédentes en utilisant un bouton début et suivant.

## 4.2 Cas d'utilisation et base de données

Quasiment tous les cas d'utilisation correspondent à une transaction à la base de données.

La seul exception est en fait lors de la phase de jeu. Lors de celle-ci, on décide de commencer une transaction lorsque l'utilisateur commence son tour et de la terminer lorsqu'il le valide.

Un exemple concret est détaillé dans la partie conception.

# 5 Conception de l'application

Cette section détaille l'architecture globale de l'application, contient le dictionnaire des classes principales et explique comment sont gérés concrètement les accès à la base de données.

## 5.1 Architecture de l'application

Notre application suit le modèle MVC. On dispose d'une couche modèle représentant les différentes données locales de notre application, une couche vue réalisée a l'aide de la bibliothèque Swing et une couche contrôleur pour faire le lien entre notre modèle et nos vues.

En plus de ces trois couches, on dispose d'une série de DAO permettant d'instancier les éléments du modèles dont on a besoin à partir de la base de données et pour sauvegarder nos modifications locales.

On retrouve également différents patron dans notre architecture. Ainsi, les historirques sont gérés par le patron commande. L'application ou l'annulation de ces actions se fait aussi à l'aide de deux visiteurs ...

Pour une meilleure visibilité, ce diagramme est également fourni en fichier image dans l'archive contenant ce fichier.

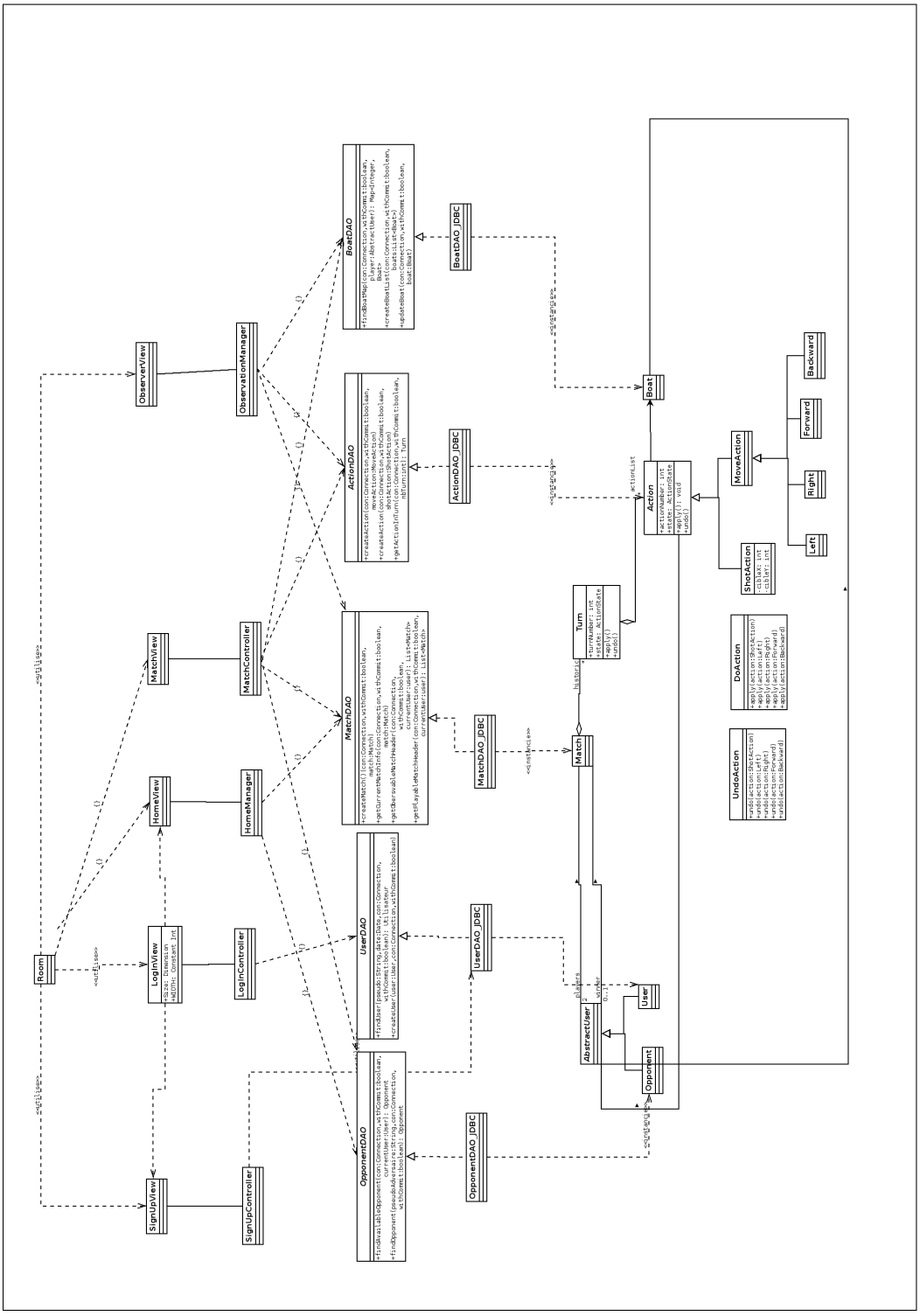


FIGURE 3 – Diagramme de classe

## 5.2 Fonctionnement de la synchronisation avec la BDD

Une partie intéressante de la conception de notre application est sa façon d'intéragir avec la base de données.

Dans le cas où l'on souhaite récupérer des données, on demande au DAO nous

intéressant de nous renvoyer la liste d'objet correspondant.

Ce qui est plus intéressant est la façon dont on va mettre à jour la BDD. En fait, on crée tout d'abord les objets en local puis on tente de les sauvegarder dans la BDD.

Si cela fonctionne, on valide la transaction et on rajoute notre objet à notre modèle. Si ce n'est pas le cas, on ne tient pas compte de cet objet et on annule l'ensemble de la transaction.

Au début d'une transaction effectuant une modification, on insère toujours un point de sauvegarde. En cas d'erreur, on reviendra dans cet état.

Par exemple, voici un diagramme de séquence montrant comment se déroule l'ajout d'un tour composé d'une action mouvement à notre partie :

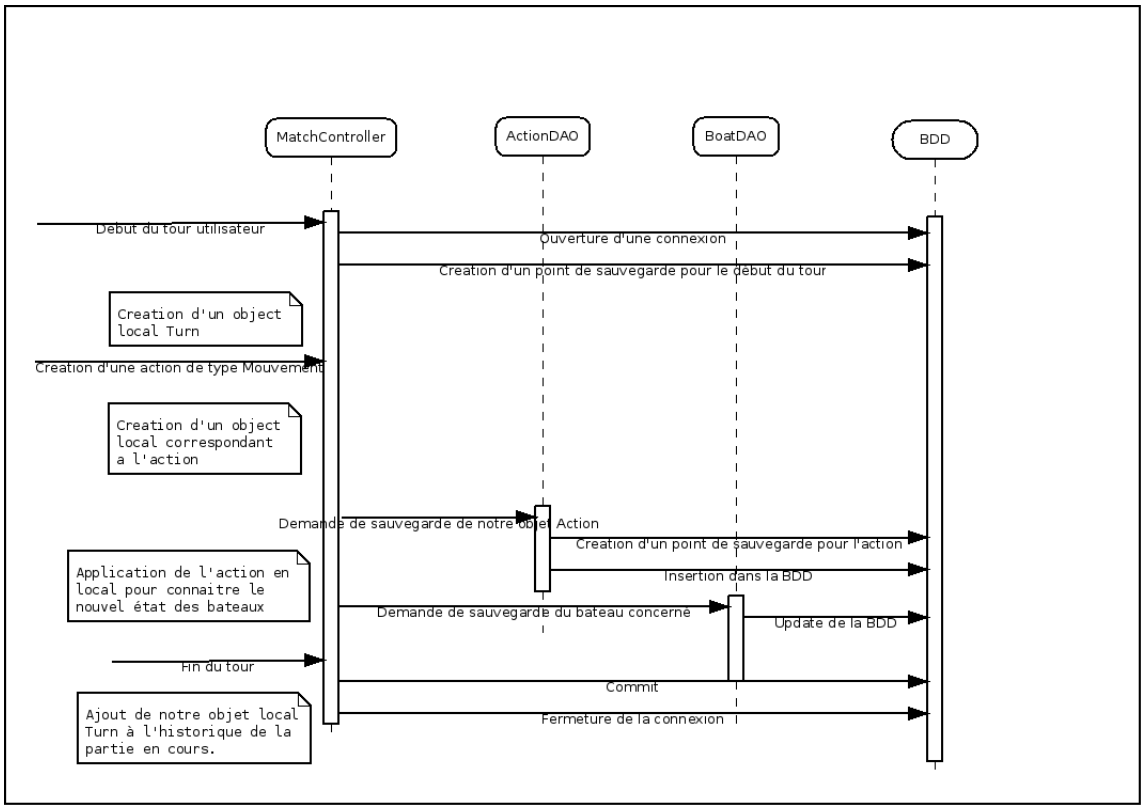


FIGURE 4 – Exemple d'une transaction sur un tour

On voit donc ici que les objets sont crée en local puis sauvegardé dans la BDD comme indiqué plus haut.

La nuance ici est qu'on insère plusieur point de sauvegarde, un pour l'ensemble du tour et un pour chaque action. En effet, si une action est incorrecte, on ne veut pas perdre les précédente. Dans le cas d'une erreur sur l'action ou si l'état des bateaux qu'elle engendre est incorrect alors on est capable de l'annuler seule.

L'action est ajoutée à notre objet tour que lorsqu'elle est dans la base de données et notre tour est lui même ajouté à notre historique que lorsque la transaction a correctement été commitée.

## 5.3 Dictionnaire de classes

Cette partie résume l'ensemble des principales classes de notre modèle et de notre couche DAO.

### 5.3.1 Le modèle

Pour la gestion des utilisateurs, nous avons les classes suivantes :

- `AbstractUser` : Super classe représentant tous les types d'utilisateur de l'application. Elle comprend un pseudo.
- `Opponent` : Sous-classe de `AbstractUser` qui représente un adversaire.
- `User` : Sous-classe de `AbstractUser` représentant un utilisateur local. Il contient l'ensemble de ses informations.

Pour la gestion des parties, nous avons les classes suivantes :

- `Boat` : Représente un bateau. Un bateau est associé à une partie et à un joueur. Il dispose en attribut de sa taille, son nombre de point de vie sa position et son orientation.
- `Position` : Représente une coordonnée dans un repère en deux dimensions.
- `Action` : Super classe représentant les actions d'un joueur. Elle dispo d'une méthode apply et undo et d'un attribut indiquant son ordre dans le tour. Elle possède une référence sur son Turn parent. Ses sous classes sont décrites un peu plus loin.
- `Turn` : Correspond à un historique d'action pour un tour. Elle dispose d'une méthode `apply` et `undo` qui applique ou annule l'ensemble de l'historique d'actions.
- `Match` : Cette classe regroupe tous les éléments d'une partie : la liste des bateaux des deux joueurs, les deux joueurs et un historique de `Turn`. Elle dispose de différente méthode pour jouer l'historique.

Pour la gestion des actions, nous utilisons différentes sous classes ainsi que deux visiteurs.

- `MoveAction` : Correspond à une action de mouvement.
- `Forward` : Correspond à l'action de mouvement avancer.
- `Backward` : Correspond à l'action de mouvement reculer.
- `Left` : Correspond à l'action de mouvement tourner à gauche.
- `Right` : Correspond à l'action de mouvement tourner à droite.

- `ShotAction` : Correspond à l'action de tir. Elle dispose d'une coordonnée cible et dans le cas où elle a touché quelque chose une fois appliquée du bateau touché.
- `DoAction` : Visiteur permetant d'appliquer n'importe quelle action.
- `UndoAction` : Visiteur permettant d'annuler n'importe quelle action.

### 5.3.2 La couche DAO

On décrit ici l'ensemble des classes abstraites DAO utilisées. Notre application fournie une implémentation de ces classes pour notre base de données en utilisant JDBC.

- `UserDAO` : Ce DAO permet de crée un utilisateur ou bien de trouver celui correspondant à un pseudo et une date de naissance (ce qui sert dans notre cas à la connexion).
- `OpponentDAO` : Ce DAO permet de récupéré un adversaire correspondant à un pseudo ou bien de nous donner un adversaire. Bien sur pour le moment la classe adversaire ne dispose que d'un pseudo mais on passe quand même par ce DAO au cas où l'on souhaite récupérer d'avantage d'information.
- `MatchDAO` : Ce DAO permet de récupéré l'ensemble des "headers" des matchs jouables ou observables. Ces headers sont composés des noms des deux joueurs participants et de la date de création. Cela permet ainsi de les lister ces matchs sans avoir à faire une copie de la BDD. Il propose aussi des méthodes pour créer une partie ou synchroniser l'état actuel de notre objet local avec la BDD.
- `ActionDAO` : Ce DAO permet de rajouter des actions à la BDD et aussi de récupérer une liste d'actions regroupée en un tour.
- `BoatDAO` : Ce DAO permet d'insérer une liste de bateaux dans la BDD ou de sauvegarder l'état courant d'un bateau.

### 5.3.3 La couche vue

La couche vue n'est pas la plus intéressante à détailler ici. Globalement, on dispose d'une vue pour la connexion, d'une pour l'inscription, d'une pour la page d'accueil, d'une pour l'observation et d'une pour le jeu.

### 5.3.4 La couche contrôleur

Chacune de nos vue crée un contrôleur lors de sa création. Celui-ci permet de gérer les actions nécessitant une intéraction avec le modèle ou la couche DAO.

Lors de l'utilisation d'une méthode d'un contrôleur qui nécessite un retour visuel, on passe la vue en paramètre.

# 6 Bilan

Les problèmes dûs à une mauvaise compréhension au début d'un projet sont les plus coûteux à corriger. Pour cette raison, nous avons tous travaillé ensemble sur l'analyse pour garantir une compréhension consistante du sujet.

Ensuite, nous nous sommes divisés en deux sous-groupes pour les étapes de conception : Fabien et Saad étaient responsables pour la conception de l'application, alors que Felipe, João Pedro et Matheus travaillaient sur la conception de la base de données. A la fin de chaque séance encadrée, les deux sous-groupes partageaient ce qu'ils avaient fait de façon à se tenir informé de l'avancée du projet et avoir une bonne notion du fonctionnement global du système. Chaque sous-groupe a aussi redigé la documentation correspondante à la partie dont il était responsable.

Finalement, nous nous sommes tous regroupés pour l'étape d'implémentation. Nous avons rencontré quelques problèmes lors de la mise en commun des différentes parties codées séparément, mais ils ont été facilement corrigés par l'entremise de la communication entre les personnes impliquées.

Quant à l'organisation des horaires de travail, même si la plupart du travail a été réaliséé lors des séances encadrées, nous avons pas mal progressé en dehors des cours.

Le développement de ce projet assez conséquent en équipe s'est rapproché de ceux de la vraie vie, de même que pour les problèmes que nous avons rencontré. Parfois, diviser le travail à faire pour un tel projet n'est pas une tâche triviale, et de ce fait, nous avons eu des difficultés pour paralléliser quelques parties de l'implémentation.